Optimising CP2K for the Intel Xeon Phi

Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Optimising CP2K for the Intel Xeon Phi

Fiona Reid[a], Iain Bethune[a]*

[a]*EPCC, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK*

Abstract

CP2K is an important European program for atomistic simulation for many users of the PRACE Research Infrastructure as well as national and local compute resources. In the context of a PRACE Preparatory Access Type C project, we have parallelised several routines in CP2K to allow the code to gain better performance on the Intel Xeon Phi for a materials science application. We have obtained a 50% speedup in the maximum performance of the code on the Xeon Phi, but have not been able to demonstrate better performance than running the same calculation on a Sandy Bridge 16-core CPU node. We present details of the developments made to CP2K, and discuss several lessons, which will be of wider interest to developers considering porting their codes to Xeon Phi.

Application Code: CP2K

1. Introduction

CP2K [1] is a freely available and widely used program for atomistic simulation in the fields of Computational Chemistry, Materials Science, Condensed Matter Physics and Biochemistry, amongst others. Today's researchers require robust and portable applications that allow them to tackle complex and challenging problems by taking advantage of the latest advances in computer hardware. CP2K has demonstrated scalability to 10,000s of CPU cores using a mixed-mode MPI/OpenMP parallelisation strategy, and has been deployed on a range of Tier-0 and Tier-1 PRACE systems. The code can make use of GPU accelerators using Nvidia's CUDA programming model, and recent work within PRACE [2] ported the code to Intel's MIC (Many Integrated Core) architecture, using the existing parallelisation, although initial performance results were disappointing.

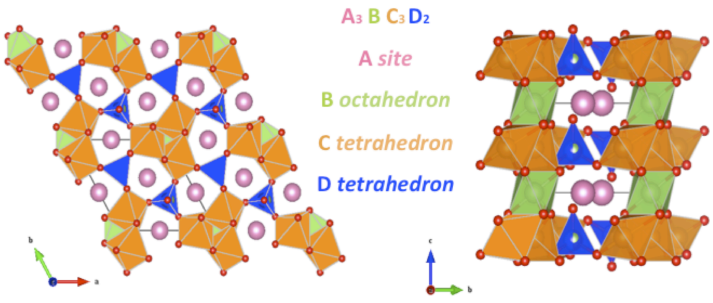

Figure 1: A generic Langasite structure that could be studied with CP2K (from Ben Slater, UCL).

* Corresponding author. *E-mail address*: ibethune@epcc.ed.ac.uk

The aim of this work was to optimise CP2K with the objective of improving the performance of a specific test simulation on the Xeon Phi. The systems of interest are the Langasites (see **Error! Reference source not found.**), a family of solid oxides composed of Lanthanum, Gallium and Germanium, studied by Dr. Ben Slater (UCL), who provided the case study input files for this project. Langasites have applications as fuel cells and we wish to be able to efficiently computationally screen different orderings of La, Ga and Ge to determine the minimum energy and maximum conductivity structures. Since many structures must be evaluated in parallel, a cluster of accelerated nodes was suggested as a suitable architecture for executing these jobs quickly and with rapid turnaround.

Hardware: EURORA

The EURORA system, located at the Cineca facility in Bologna, Italy, was used to obtain the results presented in this report.

EURORA (**EUR**opean many integrated c**OR**e **A**rchitecture) is a heterogeneous, tightly clustered Linux system running CentOS 6.3. The system is made up of 64 Intel compute nodes. Half of the compute nodes comprise two eight-core Intel Xeon E5-2658 processors running at 2.1 GHz the other half comprise two eight-core Intel Xeon E5-2678W processors running at 3.1 GHz. 58 of the nodes have 16 GB of memory but only 14 GB of this can be safely allocated by the application due to system overheads. The remaining 6 nodes (all with 3.1 GHz clock speed) have 32 GB of memory. Each node allows shared memory jobs using up to 16 threads to be run, and MPI can be used both within and between nodes. Users can specify the amount of memory and clock speed they require via the PBS batch system. In addition to the compute nodes, EURORA has a login node comprising two six-core E5645 processors running at 2.4 GHz. The processors on the login node use a different instruction set from the compute nodes - the E5645 is based on the 'Westmere' microarchitecture, while E5-2658/78W are 'Sandy Bridge' chips and support AVX. As a result binaries cross-compiled for the compute nodes cannot be run on the login nodes. This means that all testing of code must be carried out via the batch system either interactively or by remote submission.

32 of the EURORA compute nodes have two Nvidia Tesla K20 (Kepler) GPU cards attached with the remaining 32 compute nodes having two Intel Xeon Phi 5110P co-processors instead. Each Xeon Phi card contains 60 physical cores running 4 virtual threads per core giving access to a total of 240 threads per card. The clock speed of the cores is 1.053 GHz and each card has 8 GB of memory with a memory bandwidth of 352 GB/s [3]. For more details on the hardware specifications please see [4]. In the remainder of this report the Xeon E5-2678W processors will usually be referred to as the host.

The EURORA login and compute nodes have the Intel Cluster Studio XE 2013 (and 2013 SP1) software installed, which includes compilers, tools and debuggers for both the host nodes and the Xeon Phi cards. Compilation of code for the host and Xeon Phi can be carried out on either a login node or via an interactive session on one of the Xeon Phi enabled compute nodes. The advantage of compiling via an interactive session is that the code can then be easily tested on either the compute node or the Xeon Phi card. For either option the appropriate compiler modules and environment must be set up prior to compiling CP2K. Version 13.1.3 of the Intel ifort/icc compiler has been used for all the results presented in this report.

2. Initial Benchmarking

Our preliminary testing of the Langasite input files was carried out on the host so that we could experiment with different parameters, the objective being to develop a benchmark that could be run on both the host and on the Xeon Phi. The initial runtime of the benchmark on the host using 16 MPI processes was found to be over 10 hours which was simply too long and thus we needed to find some way to reduce the runtime. In addition, we also discovered that the total memory requirement of the benchmark was in excess of 20 GB with each processor using over 1.25 GB and thus the benchmark in its initial form would not be able to run on the Xeon Phi card which has only 8 GB of memory. A number of steps were taken to reduce both the runtime and the memory requirements. These steps are summarised below:

1. Reduce the maximum number of iterations for both `GEO_OPT` and `CELL_OPT` from 300 to 1. Each optimisation step is essentially identical so this has no effect on the validity of the benchmark
2. Reduce the number of SCF and outer SCF cycles from 35 and 15 respectively to 1. Each SCF cycle is identical, although reducing the number of cycles makes the some setup routines proportionally more expensive

3. Use `SZV-GTH` instead of `DZVP-GTH` as the basis set for Oxygen. This is a physical approximation, which reduces the computational cost of the calculation.
4. Reduce the value of `MGRID%CUTOFF` from 600 to 50. This reduces the size of the 3D grids used to store the plane-wave expansion of the electronic density, trading accuracy for reduced memory and time.
5. Add the `SAVE_MEM` keyword, which reduces the maximum memory by deallocating sections of the input data structure when they are no longer needed. This has no effect on the calculation, but saves a (small) amount of memory.

Steps 1 and 2 enabled the runtime to be reduced to 350 seconds on 16 MPI processes with step 3 giving a further reduction in runtime to 117 seconds. Steps 4 and 5 greatly reduced the memory requirements such that each processor required 422 MB of memory. The total memory usage does not scale directly with the number of MPI processes due to parallelisation overheads including grid halos, communication buffers etc. In practice we were able to run up to 16 MPI processes on the MIC before running out of memory.

In addition to the changes detailed above, we added the following lines to the input file:

```
&TIMINGS
  THRESHOLD 0.000001
&END
```

By default CP2K only reports timings that take more than 2% of the runtime. When parallelising the code the runtime of a routine may well decrease such that it can fall below this threshold. As a result the threshold has been decreased to 0.0001% such that we should pick up all significant timings. The final input file we used for this study can be found in Appendix A. This input file can be run on both the host and Xeon Phi using a range of MPI process counts and different numbers of threads. On the Xeon Phi memory limitations mean that not all the possible MPI process counts or numbers of threads can be executed but a good range is still possible.

Four different versions of CP2K were compiled which will subsequently be referred to as SOPT, SSMP, POPT and PSMP[b]. These correspond respectively to, a pure serial (SOPT) version, a pure OpenMP (SSMP) version, a pure MPI (POPT) version and a mixed mode MPI/OpenMP (PSMP) version. On the Xeon Phi the SOPT version was not used due to the very long runtimes that would result – parallelisation is a prerequisite to achieving good performance on the MIC architecture.

The performance of the Langasite benchmark was initially investigated on the host using a single 16-core node. All four versions of CP2K were tested. A range of thread counts or MPI process counts up to sixteen was tested for the SSMP and POPT versions respectively. For the PSMP version the number of threads was fixed e.g. 1, 2, 4 with the number of MPI processes being increased until the total number of cores used reached sixteen. In addition to this, the PSMP version was also run keeping the number of threads times the number of MPI processes fixed at sixteen to ensure that all cores on a node were used. Figure 2 shows the CP2K runtime plotted against the number of cores used for the host version of the code with Table 1 giving the actual timings. For the PSMP version using all sixteen cores the runtime is plotted against the number of OpenMP threads.

Code version	Number of MPI processes or OpenMP threads for SSMP version				
	1	2	4	8	16
SOPT	950.524	-	-	-	-
POPT	968.343	495.616	267.798	147.221	83.188
SSMP	977.185	598.108	400.982	302.575	254.448
PSMP (full node)	270.042	169.481	122.560	100.747	88.721
PSMP 1 thread	988.320	503.146	274.549	150.027	84.752
PSMP 2 threads	614.426	320.623	173.493	98.217	-
PSMP 4 threads	420.289	221.592	122.600	-	-

Table 1: Runtime (in seconds) of the Langasite benchmark run on the host (3.1GHz E5-2678W processors were used for all runs). The fastest runtime was obtained with the POPT version using 16 MPI processes.

[b] The acronyms SOPT, SSMP, POPT and PSMP are standard CP2K terminology and will be familiar to users of the code.

From Figure 2 we can see that the best performance is obtained when running the pure MPI (POPT) version of the code on 16 processors. The runtime of the OpenMP (SSMP) version is generally higher than for the POPT version as it doesn't scale as well as the MPI version. This is a simple consequence of Amdahl's law as the proportion of the code which is not fully OpenMP parallelised is non-zero. The PSMP version run on a fixed thread count can usually outperform the SSMP version as with the PSMP build the code can take advantage of both the MPI and OpenMP parallelisation. When running all sixteen cores the PSMP version gives equivalent performance to the POPT version when using 16 MPI processes. The other thread counts however never manage to do any better than the POPT version – this is because only sections of the code have been threaded and we haven't yet reached the limit of MPI scaling, where the use of OpenMP threads is expected to extend the scalability of the code [5]. The performance of the POPT version and PSMP versions with 1 thread are identical as expected. The serial (SOPT) runtime was the roughly the same as the SSMP (1 thread) or POPT (1 process) versions.

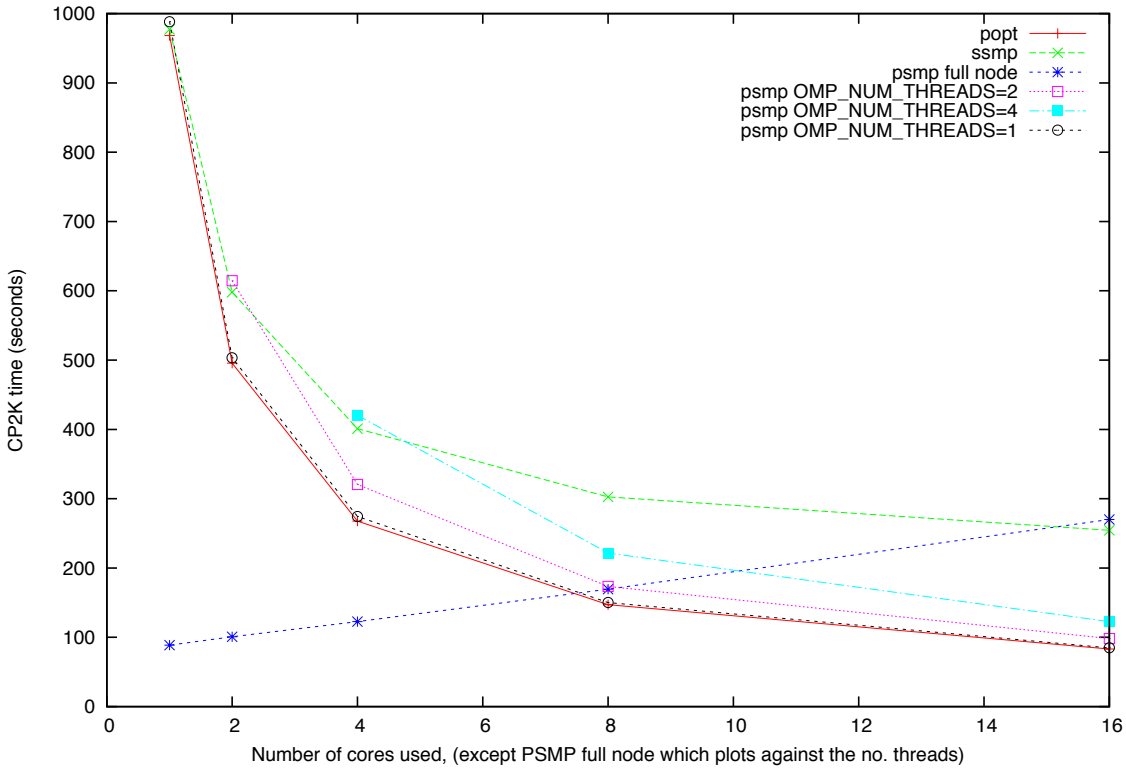

Figure 2: Performance of Langasite benchmark on a 16-processor Intel Xeon host node of EURORA.

Figure 3 gives the corresponding performance results obtained on the Xeon Phi. Some process or thread counts could not be run due to the memory limitation of the card. Comparing the performance of the host and Xeon Phi (i.e. comparing Figure 2 and Figure 3) we can see that the SSMP and POPT versions behave in a similar way. The POPT version is roughly 3 times faster than the SSMP version, which is broadly consistent with what we see for 16 threads on the host. The main difference is that on the Xeon Phi the PSMP version can outperform both the POPT and SSMP version. This is because by using OpenMP threads we are able to scale to more virtual threads/processors without exceeding the memory limitation, and thus better overall performance can be obtained.

The best performance on the Xeon Phi was obtained with the PSMP version running 8 MPI processes each with 16 OpenMP threads (this corresponds to 128 virtual threads, or a little over 50% of the available capacity of the MIC. The runtime for this test was 671 seconds. Comparing this with the fastest time obtained on the host (83 seconds on 16 MPI processes) we find that the host node is around 8 times faster than the Xeon Phi. Clearly, the code is unable to utilise the Xeon Phi to its full potential for this particular problem, since it has around 2.5x the peak FLOP/s of the Xeon E5-2678W 16-core node. The performance of CP2K on the Xeon Phi is limited for several reasons: the lack of strong scaling in some parts of the code, a potential lack of vectorisation on the Xeon Phi native build and also the increasing memory footprint with the number of threads or processes which prevents all of the MIC's resources from being utilised. In a previous PRACE white paper [2] we identified a

number of routines that were limiting the Xeon Phi performance for a different test case, although the results are similar here. Section 3 will discuss the optimisations we applied to improve the performance of CP2K on the Xeon Phi. The results and a brief discussion of the performance of the final optimised code are presented in Section 4.

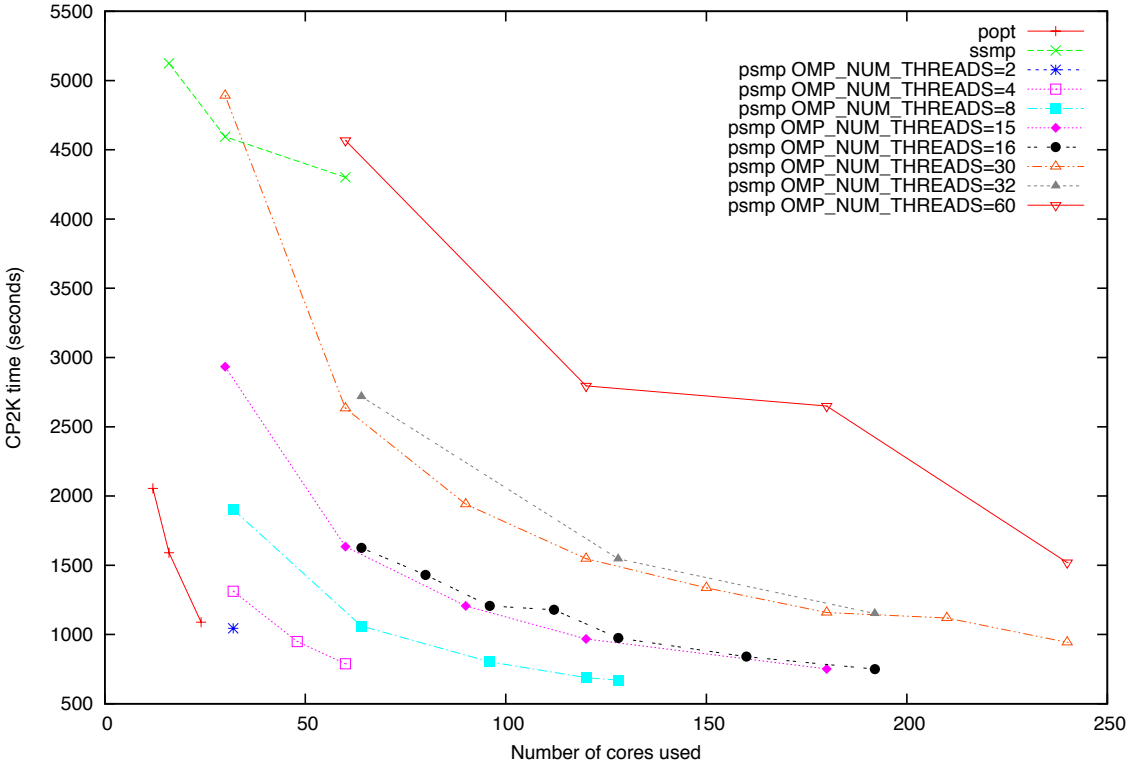

Figure 3: Performance of the Langasite benchmark on the Xeon Phi MIC card.

3. Optimisation

Subroutine `build_core_ppl`

Our initial investigations of CP2K on the Xeon Phi [2] identified a number of routines that scaled poorly. Of these routines the `build_*` routines were found to be particularly costly when larger numbers of threads were used as is required to make full use of the Xeon Phi. These `build_*` routines in CP2K are not threaded and as such the runtime remains constant with increasing thread count. Previous work [6] attempted to parallelise the `build_core_ppl` subroutine in the source file `core_ppl.F` by adding OpenMP directives to the code. `build_core_ppl` computes contributions to the core Hamiltonian matrix due to the interaction of particles via their pseudopotentials. The main change made to the code was the addition of a parallel region around a loop over all particles in the neighbour list. This approach involved the introduction of a new data structure to hold the data describing a task[c]. Prior to entering the parallel region the entire neighbour list is iterated over in serial to generate an array of tasks. A parallel region was then introduced around a new loop, which looped over the independent tasks. This approach gave good scaling on two and four threads but failed to scale beyond four threads and therefore was not added to the CP2K code base. Being able to scale this part of the calculation beyond four threads would be beneficial when running CP2K on both the Xeon Phi and also when running on modern HPC systems that typically have 16 or 32 way SMP nodes.

[c] A task is essentially one or more iterations of the main `DO WHILE` loop over particle pairs.

A second attempt to parallelise the `build_core_ppl` subroutine was also made by [6] which could utilise the iterator directly from within a parallel region. Optional arguments of the iterator can be used to specify the thread from which it is being accessed, and access elements of the list in a thread-safe manner. This approach should, in theory, be able to scale to a larger thread count as it avoids the need to pre-compute list of tasks in serial and also should have near perfect load balancing as each thread will continue to request new iterations in a dynamic manner until all the work is complete. Unfortunately, this work was not completed due to unresolved bugs and time running out. We have taken this partial implementation of the the parallel iterator as a starting point and have debugged the code such that it now works along with optimising the performance by removing unnecessary synchronisation points. As a number of parts of CP2K have a similar structure with loops over particle pairs using the neighbour list iterator, it will be possible to extend this approach to other sections of the code in future.

A summary of the main code changes that allow the parallel iterator to be used in the `build_core_ppl` subroutine is given below:

- Add new integer variables `nthread` and `mepos` to store the number of threads and the thread number respectively.
- Ensure that `omp_get_num_threads()` and `omp_get_thread_num()` are defined as integer functions.
- Add the optional `nthread` argument to the iterator creation subroutine calls to ensure that each thread has its own scratch copy of the iterator state, e.g.

  ```
  CALL neighbor_list_iterator_create(nl_iterator,sab_orb)
  ```
  Becomes:
  ```
  CALL neighbor_list_iterator_create(nl_iterator,sab_orb,nthread=nthread)
  ```
- Insert an OpenMP parallel region around the main `DO WHILE` loop over particle pairs. All variables accessed inside the parallel region are correctly scoped (as shared or private). The thread ID (`mepos`) must be determined at the beginning of the parallel region. Some array allocations needed to be moved inside the parallel region – and also be deallocated before leaving the parallel region.
- Add the `mepos` variable to the `DO WHILE` conditional e.g.

  ```
  DO WHILE (neighbor_list_iterate(nl_iterator)==0)
  ```
  Becomes:
  ```
  DO WHILE (neighbor_list_iterate(nl_iterator, mepos=mepos)==0)
  ```
- Add the optional `mepos` argument to the `get_iterator_info` call, e.g.

  ```
  CALL get_iterator_info(nl_iterator,ikind=ikind,jkind=jkind,inode=inode,&
  iatom=iatom,jatom=jatom,r=rab)
  ```
  Becomes:
  ```
  CALL
  get_iterator_info(nl_iterator,mepos=mepos,ikind=ikind,jkind=jkind,inode=inode
  ,& iatom=iatom,jatom=jatom,r=rab)
  ```
- The `get_iterator_info` routine has been modified such that it is now thread aware.
- The `neighbour_list_iterate` routine has been modified to cleanly terminate the iteration in the multi-threaded case. The original serial code allowed an extra final call to this routine, which would always exit but with multiple threads potentially able to call this routine simultaneously some additional logic was required to prevent the threads being able to over-run the end of the neighbour list data structure, producing erroneous results.
- Ensured that updates to shared variables are protected with critical sections – the `force`, `virial` and `h_block` variables require this.

Following the addition of OpenMP to the source code each version (SOPT, SSMP, POPT & SSMP) was tested for correctness by running the entire CP2K regression test suite. The test suite comprises 2450 tests, sampling most features of the code, and can be run both in parallel and serial. More details on the regression tests can be found at [7]. These tests are run before any changes are committed to the code repository to ensure that the changes produce numerically correct output.

When testing the performance of the modified code we initially examined the host performance as this could be obtained relatively quickly (recall that the host runtime was around 8 times faster than the Xeon Phi) and without waiting a significant time for jobs to pass through the queues on EURORA. Once a parallel version that passed all the regression tests had been obtained we benchmarked it on the Xeon Phi. Our Xeon Phi performance tests were carried out using the PSMP version of the code with 8 MPI processes and thread counts ranging from 1 to 30. We chose to use 8 MPI processes as this gave a reasonable runtime on one thread (~2950 seconds) and this also allowed a good range of thread counts to be tested.

Figure 4 shows a comparison of the performance before and after parallelising the `build_core_ppl` routine obtained on the Xeon Phi. It should be noted that many of the `build_*` subroutines can be invoked both with and without the forces computations as determined by the "`if(calculate_forces)`" conditionals in the CP2K code. The code timers append `_forces` when these routines have been called with force computation enabled.

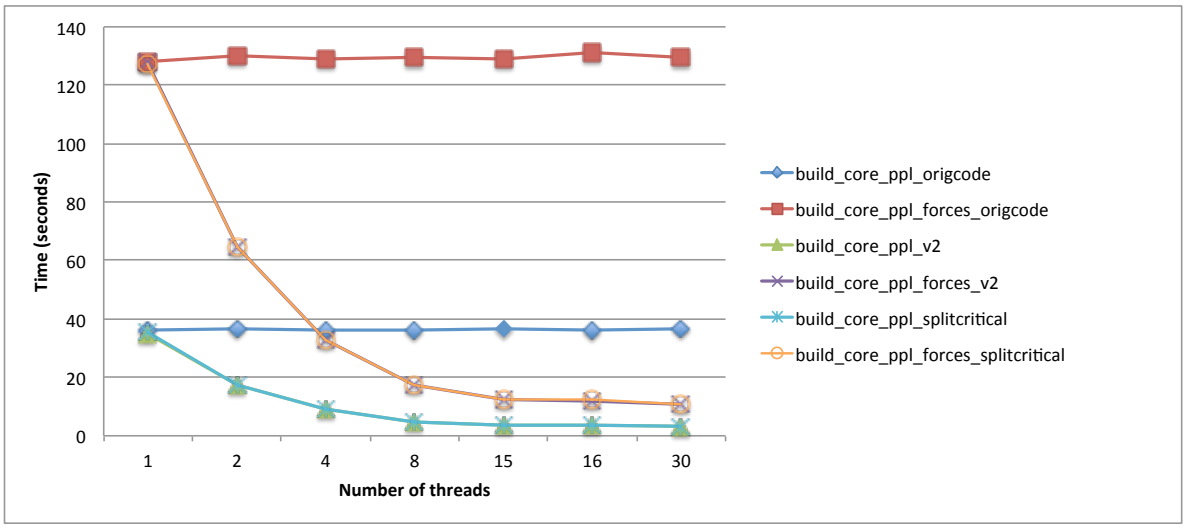

Figure 4: PSMP performance of Langasite benchmark before and after parallelising the `build_core_ppl` subroutine on Eurora Xeon Phi using 8 MPI processes.

From Figure 4 we can see that prior to parallelising the `build_core_ppl` subroutine the runtime of the `build_core_ppl` and `build_core_ppl_forces` routine is constant for all thread counts with the `build_core_ppl_forces` routine taking more than 3 times the runtime of the non forces version. After parallelisation the runtime decreases with thread counts and continues to scale up to 30 threads. The speedup of the `build_core_ppl` routines is given in Figure 5. Figure shows that the speedup of the routine involving forces is slightly better than the non-forces version since there is more computation per loop iteration, any synchronisation overhead in accessing the shared iterator or other critical sections is less significant. In both cases the code continues to speedup to 30 threads.

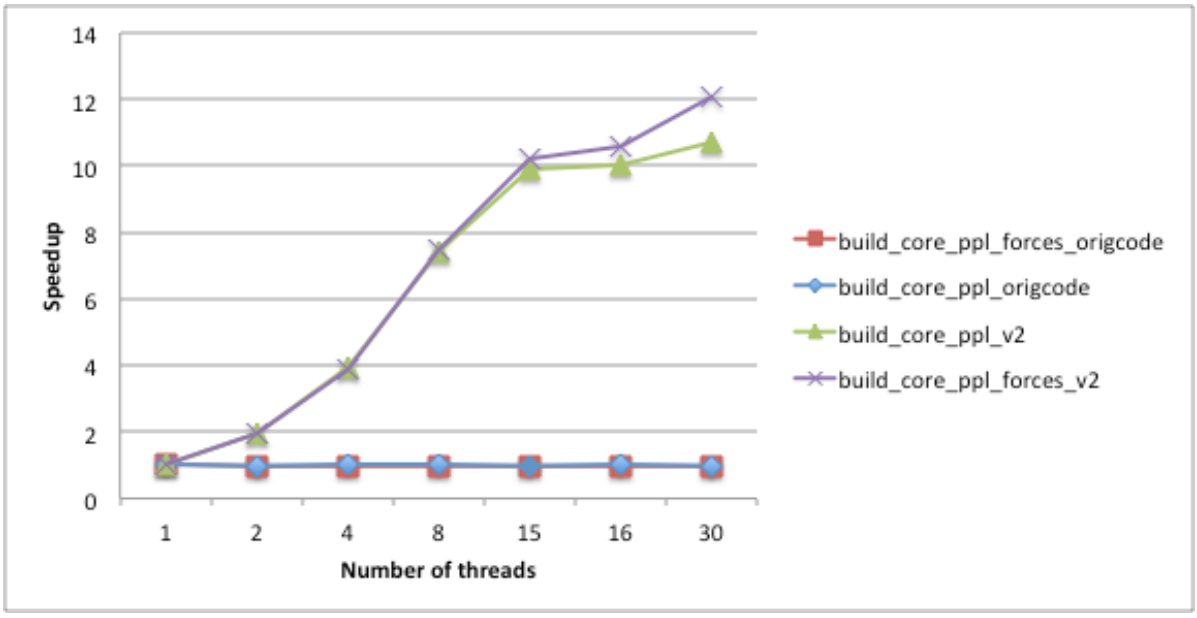

Figure 5: Speedup of the PSMP version of CP2K before and after parallelising `build_core_ppl` obtained by running the Langasite benchmark on the Xeon Phi with 8 MPI processes.

A number of ways to improve the speedup of `build_core_ppl` were investigated. The first was to split the critical regions encompassing the force and virial updates into several separate regions to investigate whether this could improve the performance. By doing this, multiple threads would update independent elements of the shared arrays at once, at the cost of increased overheads accrued from the extra critical regions. These effects seem to cancel out in practice and using several critical sections was found to give no performance benefit (see Figure 4) over having a single critical section. In the interests of code simplicity and readability a single critical section was used for each of the force and virial updates.

We also tried replacing the critical regions for the force updates with atomics as these are often faster than using critical regions in many OpenMP implementations for simple operations. However, as the data structure involved in the force updates is rather complex, accessing both array sub-sections and derived data types, it turns out that using atomics do not help. Therefore, a single critical section has been used as it improves readability of the code without any performance degradation.

Subroutine `build_core_ppnl`

A similar parallelisation method was applied to the `build_core_ppnl` subroutine in the source file `core_ppnl.F`. The `build_core_ppnl` subroutine performs a similar calculation to `build_core_ppl` (but for the non-local part of the psuedopotential) and actually has two separate `DO WHILE` loops over particle pairs using the same iterator and neighbour list data structure. The first of these loops computes the overlap integrals storing them in the `sap_int` array for use in the second loop – this means that the `sap_int` array must be shared and any updates to it protected with critical sections. At the end of the first loop the structure containing the overlap arrays `sap_int` is sorted and the second loop then computes the Hamiltonian matrix elements. In the second loop the updates to `force`, `virial` and `h_block` are protected with critical sections. Two separate parallel regions are used, one which contains the first `DO WHILE` loop and one which contains the second loop. The sorting of the integral list takes negligible time and so is not parallelised. As with the `build_core_ppl` routine, `build_core_ppnl` can be called with and without the forces computation and thus our performance results have been separated such that we can see the time spent in either calculation. Figure 6 shows the performance of the Langasite benchmark before and after the parallelisation of the `build_core_ppnl` routine. Figure 77 gives the corresponding speedup.

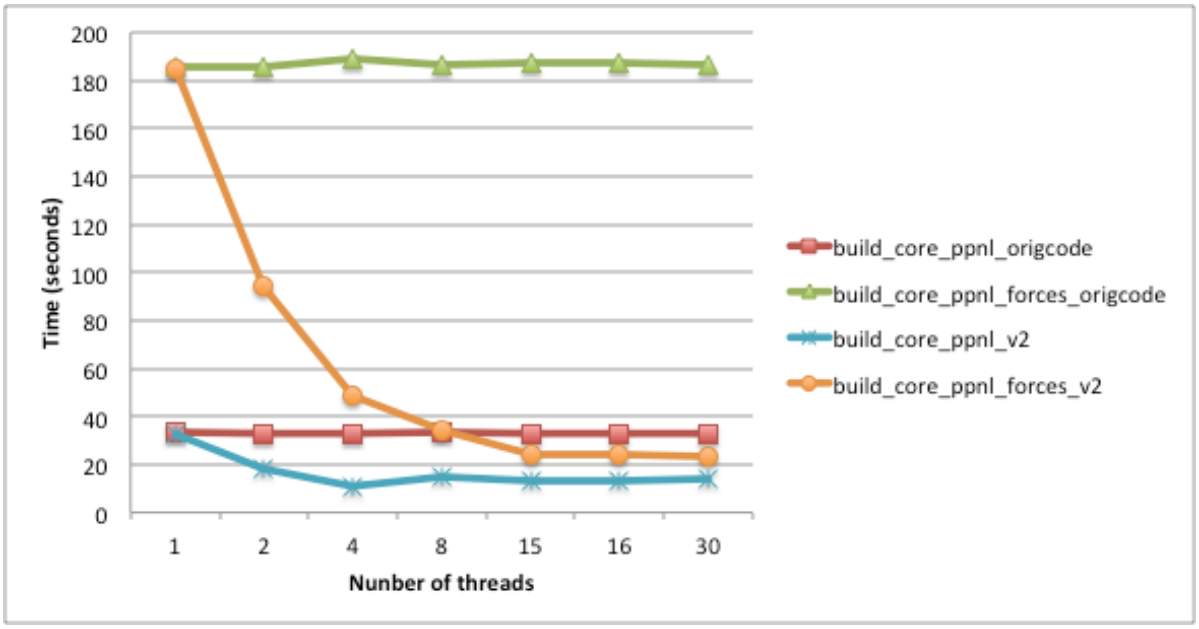

Figure 6: PSMP performance of the Langasite benchmark before and after parallelising the `build_core_ppnl` subroutine on the EURORA Xeon Phi using 8 MPI processes.

From Figure 6 and 7 we again see that prior to parallelisation the time spent in the `build_core_ppnl` routine is constant with increasing thread count. When the routine is executed with the forces computation the improvement in runtime is significant, dropping from ~185 seconds on one thread to ~23 seconds when using 30 threads – a speedup of 8. The forces computation continues to speedup even on 30 threads although as before the improvement tails off beyond 15 threads. The improvement without the forces computation is somewhat less achieving a maximum speedup of 3 on four threads, however, it still gives a small improvement in the overall runtime.

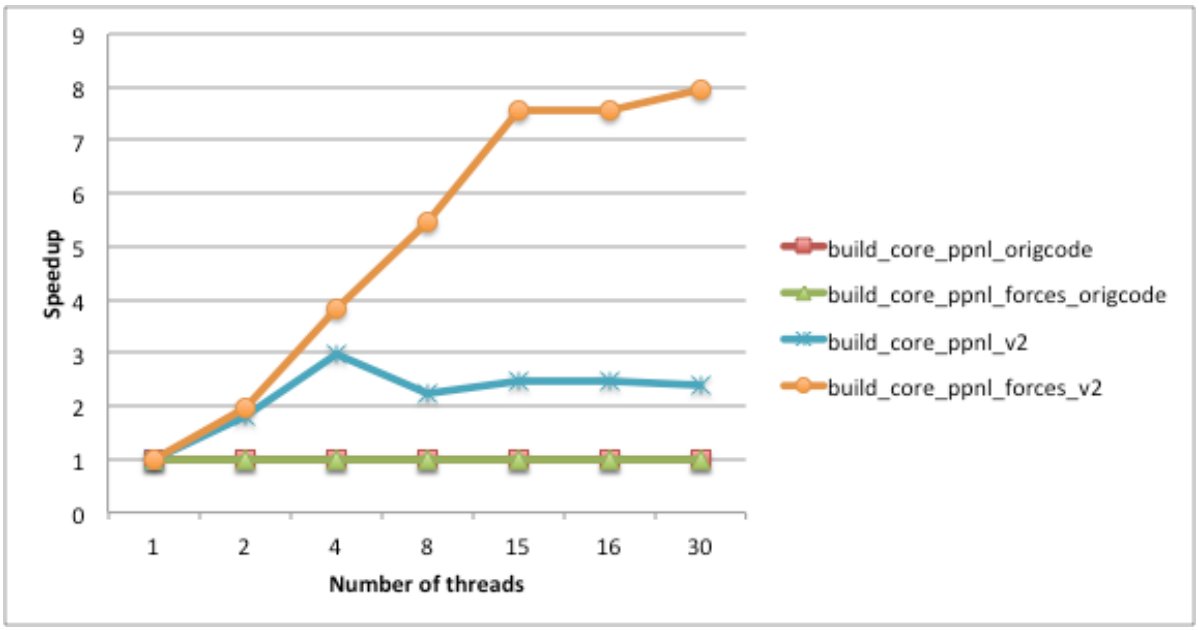

Figure 7: PSMP speedup of the Langasite benchmark before and after parallelising the `build_core_ppnl` subroutine on the EURORA Xeon Phi using 8 MPI processes.

To investigate the reasons behind the poor speedup we added some extra timers to the code such that we could compute the time spent in each parallel region and also sub-divided this into forces and non-forces computations.

The amount of computation carried out in the first loop is typically small and we wanted to make sure that our parallelisation was not introducing unnecessary OpenMP overheads. We carried out this test on the host (the host was used due to its shorter runtimes thus faster queue turn-around) with results being given by Figure 8.

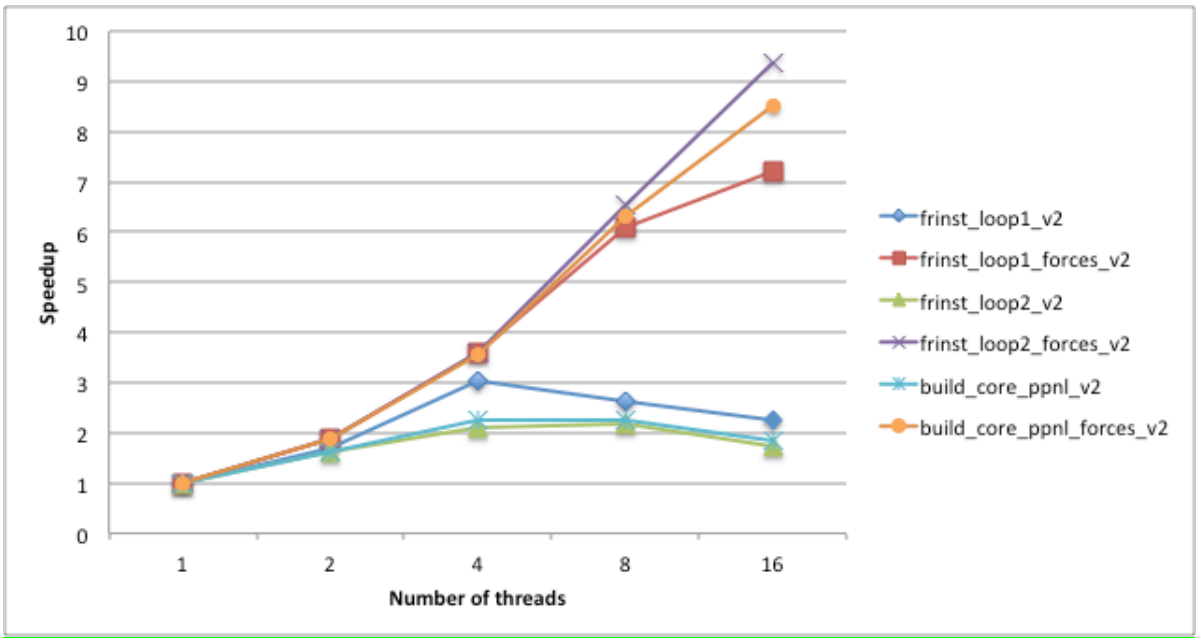

Figure 8: Performance of the SSMP version measured using the Langasite benchmark run on a single host node of EURORA. Additional timers have been added around each DO WHILE loop and these are split into forces and non-forces computations.

From Figure 8 it is apparent that as with the Xeon Phi, the non-forces subroutine doesn't scale well for either loop (blue diamonds and green triangles). However parallelisation of either loop does not seem to be introducing unnecessary overheads as the speedup is always greater than 1 and thus the final code includes both parallel regions.

Fast Fourier Transforms

In [2] we identified a bug in the FFTW3 interface of the Intel MKL library. Essentially, the FFT execute functions are not by default thread-safe which is the case for the FFTW3 library. The MKL documentation at [7] gives more detail on this and provides a solution for C which allows the user to set the number of threads which will concurrently execute a plan. We requested equivalent support for thread-safe use of the FFTW3 Fortran interface (Intel feature request ID DPD200243422), and this was added to MKL 11.1.0 (Sept 2013) meaning that we can now use the Intel MKL implementation of FFTW3 for the PSMP and SSMP versions of CP2K. The benchmarking carried out in [2] demonstrated that using MKL instead of FFTW 3.3.3 on Xeon Phi in particular should give improved performance as the Intel implementation has been optimised specifically for the Xeon Phi whereas FFTW 3.3.3 has not.

Figure 9 shows a performance comparison of CP2K's FFT routines using MKL 11.1.0 and FFTW 3.3.3 on the Xeon Phi. The PSMP version of CP2K was used for this test with the number of MPI processes fixed at 2. The benchmark is a simple input file (see Appendix B) that enables the FFT parts of CP2K to be tested in isolation from other parts of the code. It can be used for debugging, benchmarking and also is one of the regression tests that the code must pass after making changes to the source. The benchmark uses a 125 x 125 x 125 element grid, which is slightly bigger than that used by the Langasite benchmark, but is typical of a wide range of CP2K jobs. From Figure 9 we can see that MKL clearly outperforms FFTW 3.3.3 up to 32 threads beyond which it doesn't really make any difference what FFT library is used. The initial benchmarking on the Xeon Phi uses FFTW 3.3.3

The final optimised runs of the Langasite benchmark were performed using CP2K compiled with MKL 11.1.0. However, the Langasite benchmark spends a tiny (<1% of the total runtime for the host and Xeon Phi) of its runtime in FFT calculations and so using MKL 11.1.0 does not impact significantly on performance. However, other CP2K calculations can spend significant time performing FFT computations and therefore it is advisable to use MKL 11.1.0 if possible instead of the reference FFTW 3 library.

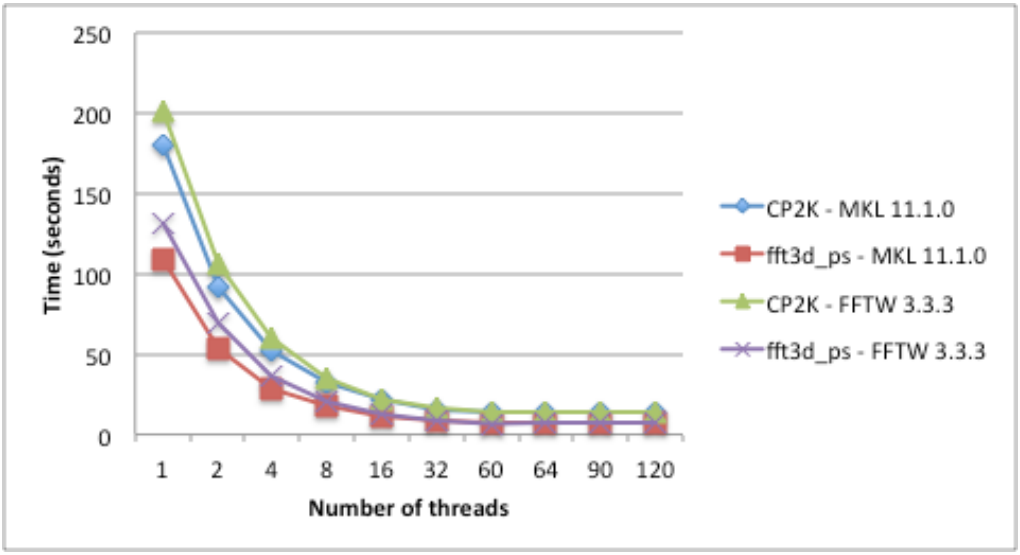

Figure 9: Performance of the PSMP version of CP2K using 2 MPI processes on the Xeon Phi using the FFT benchmark.

Additional improvements

In addition to the optimisations described above a number of other improvements were made to the CP2K source code. Some compiler related problems or bugs were also identified. Each improvement or outstanding issue is described briefly below.

1. The `rs_distribute_matrix` routine was optimised such that the local threaded data movement was overlapped with the `MPI_Alltoall` call. In making this change the allocation and initialisation of an array of locks (prerequisite to the threaded work) was moved such that it occurs before the `MPI_Alltoall` call. This allows the master thread to call MPI, while the other threads proceed with some independent work whilst the MPI communication is taking place. The original code had waited for this MPI call to complete before creating the array of locks. The deallocation of the array of locks has also been moved inside the parallel region and is now enclosed within a single directive with the `NOWAIT` clause as only one thread needs to deallocate this array. The remaining threads can continue whilst this deallocation is taking place.

2. Following addition of the `swarm_methods.F` source file (SVN revision 13372) it was discovered that the Intel compiler cannot handle compound format statements and as a result failed to compile the code. Statements of the form:

   ```
   write(unit,"(AI10)") "value: ", entry%value_i4
   ```

   which would print a some text followed by an integer of field width 10 must be re-written with a comma separating the two fields e.g.

   ```
   write(unit,"(A,I10)") "value: ", entry%value_i4
   ```

   After making such changes to the `swarm_methods.F` source file (SVN revision 13374) the code compiles and runs successfully with the Intel compiler suite.

3. Fixed a problem with argument ordering which was picked up by ifort (SVN revision 13091). The variable `stack_size` was declared after it was used in the source files: `dbcsr_mm_hostdrv_d.F`, `dbcsr_mm_hostdrv_z.F`, `dbcsr_mm_hostdrv_s.F` and `dbcsr_mm_hostdrv_c.F`. E.g. the code initially had syntax of the form:

   ```
   INTEGER, DIMENSION(dbcsr_ps_width,1:stack_size), INTENT(IN) :: params
   INTEGER, INTENT(IN) :: stack_size
   ```

   Which when compiled with ifort results in a compiler error. The correct code is to declare `stack_size` before it is used, e.g.

```
INTEGER, INTENT(IN) :: stack_size
INTEGER, DIMENSION(dbcsr_ps_width,1:stack_size), INTENT(IN) :: params
```

4. The Intel compiler loses track of the optional argument `blk_p` in `dbcsr_sort_indices` when the routine was called from `transpose_index_local`. This has been submitted to Intel as a compiler bug and is still awaiting a resolution. A workaround was subsequently supplied by Intel, which involves providing the upper bound of the arrays. This workaround has been included in CP2K SVN revision 13197.
5. The PSMP version of the code cannot currently be run with ifort 14.0.0. The code crashes with a segmentation fault, which has been confirmed by Intel as a compiler bug. Unfortunately there is no work around at present and users therefore will need to wait until the next release of the Intel compiler before a fix will be available. Older versions of the compiler e.g. 13.1.3 and 13.1.0 are unaffected by this bug.

4. Results

After optimising the code as described in section 3 we re-ran our initial set of benchmarks using the different parallel versions of CP2K. Figure 10 shows the final performance of CP2K on a 3.1 GHz host node. The overall shape of the graph is similar to our original results. The POPT and SOPT runtimes are almost identical to our original results (c.f. Figure 2 and Table 1) as would be expected as the serial and MPI parts of the code have not been affected by the optimisations we applied. The runtimes of the SSMP and PSMP versions of the code have decreased such that the SSMP version is now up to 29% (on 16 threads) faster and the PSMP version up to 27% faster (16 threads, 1 MPI process) than the original code. The best performance on the host was obtained with the POPT version using 16 MPI processes with a runtime of 83s. This is expected, as when using a single host node the POPT version of the code is the most efficient choice as we are still able to scale well with MPI. The SSMP and PSMP runtimes, although better than the initial version of the code are still greater than those obtained with the POPT version.

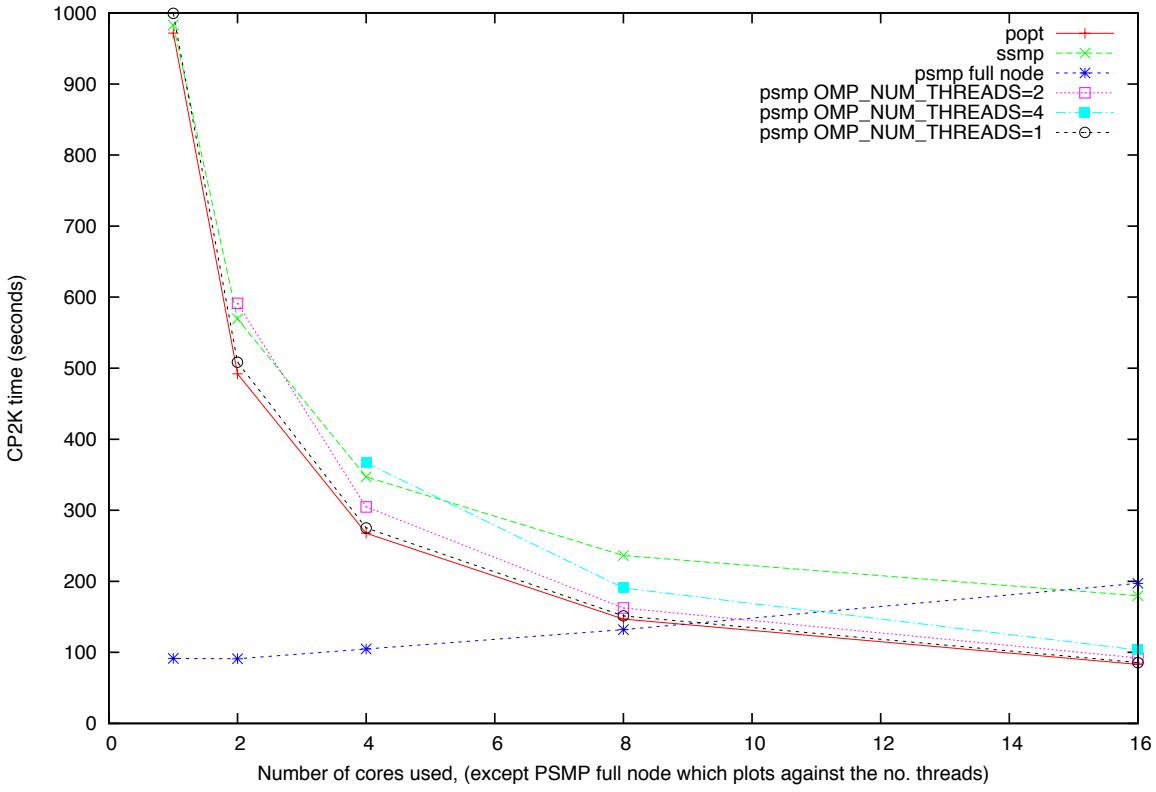

Figure 10: Final performance of Langasite benchmark on a 16 processor host node of EURORA.

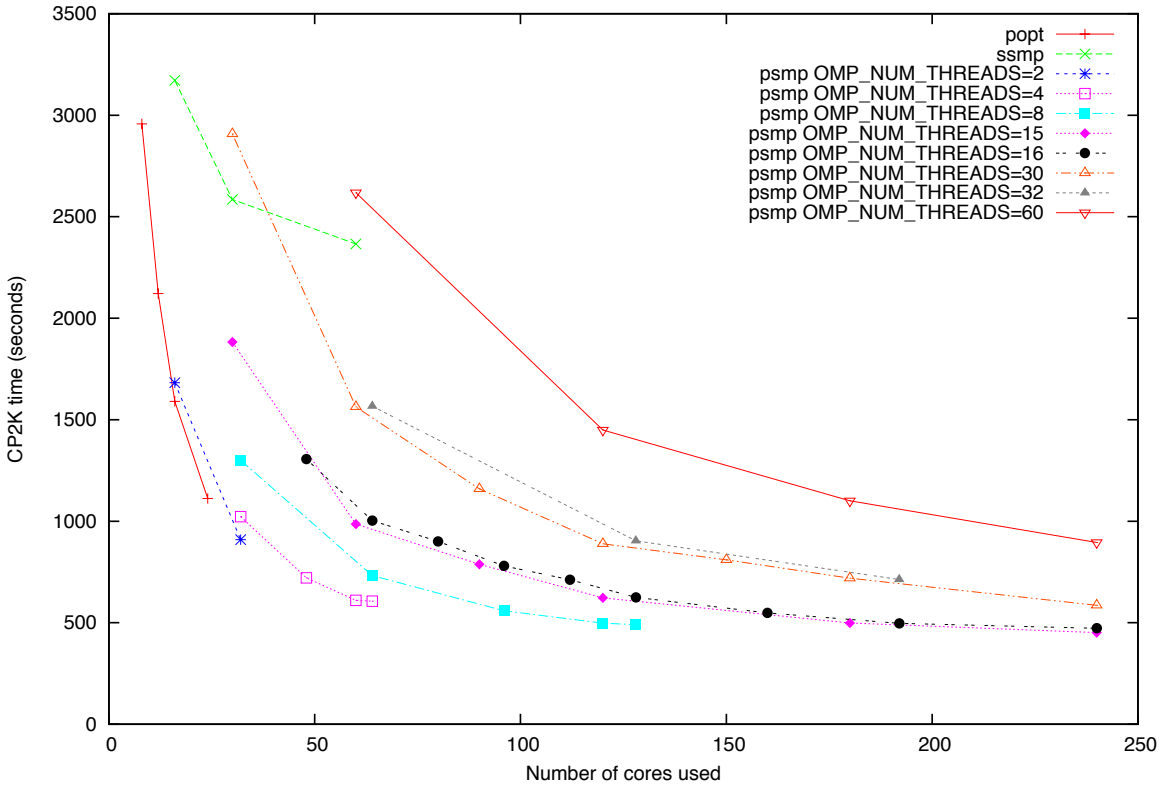

Figure 11: Final performance of the Langastite benchmark on the Xeon Phi.

Error! Reference source not found.1 shows the performance of the Langasite benchmark obtained by running the final optimised code on the Xeon Phi. As with the host version, the runtime of the POPT version is largely unchanged. However, the runtime of the SSMP and PSMP versions have decreased by up to 49% (using SSMP 60 threads) and 58% (using PSMP 3 MPI processes running 60 threads) respectively. With the optimised code, the best runtime on the Xeon Phi was 451 seconds obtained with the PSMP version using 16 MPI processes each running 15 OpenMP threads. Comparing this against the best host runtime we now get a ratio of 5.43, which is a significant improvement relative to the original code (where the Xeon Phi was 8 times slower). The Xeon Phi performance is still much lower than the host, although we have not yet been able to realise all the optimisations mentioned in [2].

As a final comparison we include the final profiles as obtained from CP2K's in built timers for the best performing host and Xeon Phi configurations. Figure 12 shows the timing profile for the POPT version run on 16 MPI processes on the host whereas Figure 13 shows the timing profile for the PSMP version run on 16 MPI processes with 15 OpenMP threads on the Xeon Phi. In each profile only sections of code taking more than 5% of the total runtime are included.

SUBROUTINE	CALLS	ASD	SELF TIME		TOTAL TIME	
			AVERAGE	MAXIMUM	AVERAGE	MAXIMUM
CP2K	1	1.0	0.086	0.109	83.146	83.149
cp_cell_opt	1	2.0	0.010	0.019	81.828	81.832
geoopt_bfgs	1	3.0	0.092	0.128	81.746	81.751
cp_eval_at	2	4.0	0.017	0.026	81.204	81.207
qs_forces	2	5.0	0.025	0.037	81.052	81.059
qs_energies_scf	2	6.0	0.013	0.021	53.872	53.893
scf_env_do_scf	2	7.0	0.004	0.007	39.986	39.986
qs_ks_build_kohn_sham_matrix	6	9.0	0.015	0.020	29.521	29.536
sum_up_and_integrate	6	10.0	0.002	0.003	26.636	27.786
integrate_v_rspace	6	11.0	26.567	27.719	26.634	27.785
init_scf_loop	4	8.0	0.001	0.003	23.455	23.460
qs_rho_update_rho	6	9.2	0.001	0.001	17.238	17.278
calculate_rho_elec	6	10.2	16.366	17.014	17.238	17.277

qs_ks_update_qs_env_forces	2	6.0	0.000	0.000	17.176	17.182
scf_env_do_scf_inner_loop	4	8.0	0.002	0.005	16.492	16.504
cp_dbcsr_multiply_d	112	12.2	0.002	0.003	14.048	15.690
dbcsr_mm_cannon_multiply	112	14.2	0.098	0.173	14.035	15.675
cp_dbcsr_mult_NS_NR	44	13.2	0.003	0.006	10.951	12.596
qs_ks_update_qs_env	8	9.0	0.000	0.000	12.362	12.372
cannon_multiply_low	112	15.2	0.069	0.088	11.371	11.543
cannon_multiply_low_multrec	448	16.2	0.002	0.003	9.856	10.214
cannon_multiply_low_multrec_mas	448	17.2	9.850	10.210	9.850	10.210
build_core_hamiltonian_matrix_f	2	6.0	0.000	0.000	9.297	9.960
prepare_preconditioner	4	9.0	0.000	0.000	9.347	9.359
init_scf_run	2	7.0	0.003	0.008	7.766	7.769
scf_env_initial_rho_setup	2	8.0	0.003	0.008	7.763	7.768
make_preconditioner	4	10.0	0.000	0.000	6.955	6.959
qs_energies_init_hamiltonians	2	7.0	0.008	0.014	5.613	5.621
build_core_ppnl_forces	2	7.0	5.032	5.587	5.032	5.587
qs_scf_loop_do_ot	4	9.0	0.000	0.000	4.761	4.792
subspace_eigenvalues_ks_dbcsr	8	10.5	0.000	0.000	4.393	4.406
ot_scf_mini	4	10.0	0.003	0.006	4.317	4.321

Figure 12: Timing profile of the POPT version using 16 MPI processes on the host. Only timings taking more 5% of the total runtime are included.

SUBROUTINE	CALLS	ASD	SELF TIME		TOTAL TIME	
			AVERAGE	MAXIMUM	AVERAGE	MAXIMUM
CP2K	1	1.0	0.545	0.611	451.241	451.294
cp_cell_opt	1	2.0	0.001	0.002	445.632	445.856
geoopt_bfgs	1	3.0	1.273	1.955	444.898	445.122
cp_eval_at	2	4.0	0.327	0.331	438.415	438.416
qs_forces	2	5.0	0.346	0.558	436.583	436.583
qs_energies_scf	2	6.0	0.008	0.022	302.840	303.049
scf_env_do_scf	2	7.0	0.001	0.005	191.614	191.942
init_scf_loop	4	8.0	0.001	0.001	126.739	126.752
qs_ks_build_kohn_sham_matrix	6	9.0	0.204	0.287	96.419	97.163
sum_up_and_integrate	6	10.0	0.026	0.028	83.097	87.559
integrate_v_rspace	6	11.0	82.344	86.813	83.071	87.531
prepare_preconditioner	4	9.0	0.000	0.000	80.328	80.789
qs_energies_init_hamiltonians	2	7.0	0.000	0.001	78.848	78.848
build_core_hamiltonian_matrix_f	2	6.0	0.013	0.122	67.793	72.663
scf_env_do_scf_inner_loop	4	8.0	0.002	0.007	62.241	64.678
make_preconditioner	4	10.0	0.001	0.001	61.477	61.551
qs_ks_update_qs_env_forces	2	6.0	0.000	0.000	59.688	59.933
cp_dbcsr_multiply_d	112	12.2	0.006	0.007	54.521	59.743
dbcsr_mm_cannon_multiply	112	14.2	2.550	4.719	54.376	59.607
qs_rho_update_rho	6	9.2	0.000	0.001	50.879	51.059
calculate_rho_elec	6	10.2	47.032	49.578	50.878	51.058
cp_dbcsr_mult_NS_NR	44	13.2	0.002	0.003	42.662	48.836
calculate_dispersion_pairpot	2	8.0	46.561	46.561	46.561	46.561
cannon_multiply_low	112	15.2	0.788	0.960	40.393	45.717
cannon_multiply_low_multrec	448	16.2	32.617	38.201	38.540	43.896
cp_fm_syevd	14	11.7	0.001	0.001	40.739	40.997
cp_fm_syevd_base	14	12.7	40.738	40.996	40.738	40.996
qs_ks_update_qs_env	8	9.0	0.000	0.000	37.193	37.674
cp_dbcsr_syevd	12	12.0	0.053	0.117	37.243	37.305
subspace_eigenvalues_ks_dbcsr	8	10.5	0.002	0.003	36.346	36.858
build_kinetic_matrix	4	8.0	29.351	34.608	29.382	34.639
build_core_hamiltonian_matrix	2	8.0	0.001	0.001	28.593	28.593
qs_scf_loop_do_ot	4	9.0	0.000	0.000	27.564	28.032
init_scf_run	2	7.0	0.001	0.005	27.996	28.001
scf_env_initial_rho_setup	2	8.0	0.001	0.002	27.995	28.000
make_full_single_inverse	4	11.0	0.099	0.169	25.836	25.895
ot_scf_mini	4	10.0	0.003	0.003	25.769	25.801
build_core_ppnl_forces	2	7.0	22.054	25.619	22.054	25.619

Figure 13: Timing profile of the PSMP version using 16 MPI processes and 15 OpenMP threads on the Xeon Phi. Only timings taking more than 5% of the total runtime are included.

Comparing the two profiles we can see that the next targets for optimisation on the Xeon Phi would be `calculate_dispersion_pairpot` and `build_kinetic_matrix`, both of which have similar structures to the `build_core_ppl` and `build_core_ppnl` which were optimised during this project. To achieve competitive performance in `integrate_v_rspace` and `calculate_rho_elec` (the two most expensive routines on the host), would require significant memory reductions to allow use of all 240 virtual threads on the MIC, which could not be achieved within the short duration of this project.

5. Lessons Learned and Recommendations

After initially porting then optimising CP2K on the Intel Xeon Phi, we have learned several important lessons which may be of interest to other developers considering porting their codes to Xeon Phi.

- **Porting:** Porting an existing parallel code is very straightforward as Intel support a wide range of programming models, including the widely use MPI and OpenMP as well as Intel-specific models like Cilk+ and TBB, among others. If the code can already be reliably compiled with the Intel compiler suite and MKL this is ideal, as most of the problems we discovered in porting were related to the Intel toolchain rather than the Xeon Phi Architecture
- **Native mode:** If the existing code has been designed for fat multi-core nodes typically found in modern HPC architectures, native mode appears to be an attractive option for utilising the Xeon Phi. However, the dual requirements for low memory usage and high scalability for a correspondingly small problem size mean that a code which has been designed and optimised for the usual 10s of CPU cores with around 1GB of memory per core, will struggle to adapt well to the Xeon Phi without major modifications. To find the required levels of concurrency (240 threads of execution) requires much finer-grained parallelism, more typical of the extreme data-parallelism used when porting applications to GPU.
- **Offload mode:** While we did not test offload mode in this project, given our experience it seems as though this is a more suitable mode of operation for several reasons. Firstly, only the parts of the code which exhibit high levels of parallelism can be executed on the Xeon Phi, for example using OpenMP to manage the threaded execution of an offload region in the code. This in turn reduces the memory requirement, since only the relevant data structures need to be copied to the co-processor. In addition, since MPI can also be used to couple together many host nodes (each with Xeon Phi co-processors), in a distributed memory code, this could reduce the size of the data structures still further and allow scaling to much larger problem sizes.
- **Extreme parallelism:** Intel's development environment for the Xeon Phi emphasises ease of use (by supporting familiar programming models). However, the need to expose high levels of parallelism is still the most demanding task, irrespective of the programming model, and developers considering porting a code should be aware of this.

6. Conclusion

Our work to optimise the performance of CP2K for simulations of Langasites on the Intel Xeon Phi in native mode has resulted in a total speedup of 49% compared with the initial version of the code. Despite this, it is still faster to perform these calculations solely on the host CPU. In addition, we were unable to run the calculation at full accuracy on the Xeon Phi due to memory constraints. As a result, we do not recommend proceeding with these calculations on the Xeon Phi, without further work on the code first to make use of offload mode to run only the most computationally intensive and potentially parallel parts of the calculation on the co-processor.

Nevertheless, our improvements to the code - the OpenMP parallelisation of two new routines, optimised FFTs using MKL, and other code quality changes – will benefit any users of the code, even on standard CPUs. These changes are already included in the latest SVN trunk and will form part of the CP2K 2.5 release during 2014.

References

[1] CP2K: Open Source Molecular Dynamics, http://www.cp2k.org
[2] Evaluating CP2K on Exascale Hardware: Intel Xeon Phi, PRACE Whitepaper, F. Reid and I. Bethune, 2013.
[3] A Quick Guide of Intel MIC Usage; http://www.hpc.cineca.it/content/quick-guide-intel-mic-usage
[4] Intel Xeon Phi Specifications; http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html
[5] Improving the scalability of CP2K on multi-core systems, I. Bethune, HECToR dCSE Report, Sep 2010

[6] CP2K – Scalable Atomistic Simulation for the PRACE Community, I. Bethune, A. Carter, K. Stratford and P. Korosoglou, 2011; http://www.prace-ri.eu/IMG/pdf/CP2K_-_Scalable_Atomistic_Simulation_for_the_PRACE_Community.pdf
[7] CP2K Regression Tester webpage; http://people.web.psi.ch/krack/cp2k/regtest/regtest.html
[8] Intel documentation regarding using FFTW3 wrappers in MKL; http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/GUID-D17B3AB4-BD4E-4652-94A7-BAD4130CCB4A.htm

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-261557.

Appendix A: CP2K input file for Langasite benchmark

This appendix contains the CP2K input file for running the Langasite benchmark on the host and Xeon Phi. Supporting files (basis sets, psuedopotentials, and dispersion potential parameters) can be found in the CP2K source distribution.

```
# CP2K input file for Langasite benchmark

&MOTION
  &GEO_OPT
    MAX_ITER 1
    MINIMIZER BFGS
  &END
  &CELL_OPT
    TYPE DIRECT_CELL_OPT
    OPTIMIZER BFGS
    MAX_ITER 1
    EXTERNAL_PRESSURE [bar] 1.0
  &END
&END
&FORCE_EVAL
  METHOD QS
  STRESS_TENSOR ANALYTICAL
  &DFT
    BASIS_SET_FILE_NAME ./BASIS_MOLOPT
    BASIS_SET_FILE_NAME ./GTH_BASIS_SETS
    POTENTIAL_FILE_NAME ./GTH_POTENTIALS
    &MGRID
      CUTOFF 50
    &END MGRID
    &QS
      EPS_DEFAULT 1.0E-12
    &END QS
    &SCF
      SCF_GUESS ATOMIC
      EPS_SCF 1.0E-7
      &OT ON
        PRECONDITIONER FULL_SINGLE_INVERSE
      &END OT
      MAX_SCF      1
      &OUTER_SCF
          EPS_SCF 1.0E-7
          MAX_SCF 1
      &END
    &END SCF
    &XC
      &XC_FUNCTIONAL PBE
      &END
      &vdW_POTENTIAL
         DISPERSION_FUNCTIONAL PAIR_POTENTIAL
         &PAIR_POTENTIAL
            TYPE DFTD3
            PARAMETER_FILE_NAME ./dftd3.dat
            REFERENCE_FUNCTIONAL PBE
            R_CUTOFF 15.
         &END PAIR_POTENTIAL
      &END vdW_POTENTIAL
    &END XC
  &END DFT
  &SUBSYS
    &CELL
      ABC               16.449200 16.449600 15.471100
      ALPHA_BETA_GAMMA -89.990000 90.000000 120.000000
      PERIODIC XYZ
    &END CELL
    &COORD
```

```
O       0.000756    4.748306    0.946831
O       4.111444    2.375618    4.208139
O       2.450075    2.355151    1.612089
O       4.960243    0.944820    1.613636
O       4.924837    3.823890    1.612089
O       0.500832    3.253009    3.468621
O      -1.544651    5.063646    3.467073
O       1.044441    5.929789    3.467073
O       1.426135    0.770091    4.065805
O      6.844500    0.849868    4.067352
O      4.068632    5.503961    4.067352
O     -0.481982    1.464744    1.381569
O     -2.603181    6.808332    1.380022
O      3.084964    5.973530    1.381569
O      4.112956   11.873271   11.261413
O      8.223644    9.500583   14.522721
O      6.562275    9.480115   11.926671
O      9.072443    8.069784   11.926671
O      9.037037   10.948854   11.926671
O      4.613032   10.377973   13.781656
O      2.567549   12.188610   13.780108
O      5.156641   13.054753   13.781656
O      5.538335    7.895056   14.380387
O     10.956700    7.974832   14.381934
O      8.180832   12.628925   14.380387
O      3.630218    8.589708   11.694604
O      1.509019   13.933297   11.694604
O      7.197164   13.098495   11.696151
O     -4.111644   11.873271   11.261413
O     -0.000956    9.500583   14.522721
O     -1.662325    9.480115   11.926671
O      0.847843    8.069784   11.926671
O      0.812437   10.948854   11.926671
O     -3.611568   10.377973   13.781656
O     -5.657051   12.188610   13.780108
O     -3.067959   13.054753   13.781656
O     -2.686265    7.895056   14.380387
O      2.732100    7.974832   14.381934
O     -0.043768   12.628925   14.380387
O     -4.594382    8.589708   11.694604
O     -6.715581   13.933297   11.694604
O     -1.027436   13.098495   11.696151
O      8.225356    4.750385   11.261413
O     12.336044    2.377697   14.522721
O     10.674675    2.357230   11.926671
O     13.184843    0.946898   11.926671
O     13.149437    3.825969   11.926671
O      8.725432    3.255087   13.781656
O      6.679949    5.065724   13.780108
O      9.269041    5.931868   13.781656
O      9.650735    0.772170   14.380387
O     15.069100    0.851946   14.381934
O     12.293232    5.506040   14.380387
O      7.742618    1.466822   11.694604
O      5.621419    6.810411   11.694604
O     11.309564    5.975609   11.696151
O      0.000756    4.750385   11.261413
O      4.111444    2.377697   14.522721
O      2.450075    2.357230   11.926671
O      4.960243    0.946898   11.926671
O      4.924837    3.825969   11.926671
O      0.500832    3.255087   13.781656
O     -1.544651    5.065724   13.780108
O      1.044441    5.931868   13.781656
O      1.426135    0.772170   14.380387
O      6.844500    0.851946   14.381934
O      4.068632    5.506040   14.380387
O     -0.481982    1.466822   11.694604
O     -2.603181    6.810411   11.694604
```

```
O      3.084964    5.975609   11.696151
O      4.112956   11.872232    6.104896
O      8.223644    9.499543    9.366204
O      6.562275    9.479076    6.768606
O      9.072443    8.068745    6.770153
O      9.037037   10.947815    6.768606
O      4.613032   10.376934    8.625138
O      2.567549   12.187571    8.623591
O      5.156641   13.053714    8.625138
O      5.538335    7.894016    9.223870
O     10.956700    7.973793    9.223870
O      8.180832   12.627886    9.223870
O      3.630218    8.588669    6.538087
O      1.509019   13.932258    6.538087
O      7.197164   13.097455    6.538087
O     -4.111644   11.872232    6.104896
O     -0.000956    9.499543    9.366204
O     -1.662325    9.479076    6.768606
O      0.847843    8.068745    6.770153
O      0.812437   10.947815    6.768606
O     -3.611568   10.376934    8.625138
O     -5.657051   12.187571    8.623591
O     -3.067959   13.053714    8.625138
O     -2.686265    7.894016    9.223870
O      2.732100    7.973793    9.223870
O     -0.043768   12.627886    9.223870
O     -4.594382    8.588669    6.538087
O     -6.715581   13.932258    6.538087
O     -1.027436   13.097455    6.538087
O      8.225356    4.749346    6.104896
O    12.336044    2.376658    9.366204
O    10.674675    2.356190    6.768606
O    13.184843    0.945859    6.770153
O    13.149437    3.824929    6.768606
O     8.725432    3.254048    8.625138
O     6.679949    5.064685    8.623591
O     9.269041    5.930828    8.625138
O     9.650735    0.771131    9.223870
O    15.069100    0.850907    9.223870
O    12.293232    5.505000    9.223870
O     7.742618    1.465783    6.538087
O     5.621419    6.809372    6.538087
O    11.309564    5.974570    6.538087
O     0.000756    4.749346    6.104896
O     4.111444    2.376658    9.366204
O     2.450075    2.356190    6.768606
O     4.960243    0.945859    6.770153
O     4.924837    3.824929    6.768606
O     0.500832    3.254048    8.625138
O    -1.544651    5.064685    8.623591
O     1.044441    5.930828    8.625138
O     1.426135    0.771131    9.223870
O     6.844500    0.850907    9.223870
O     4.068632    5.505000    9.223870
O    -0.481982    1.465783    6.538087
O    -2.603181    6.809372    6.538087
O     3.084964    5.974570    6.538087
O     4.112956   11.871192    0.946831
O     8.223644    9.498504    4.208139
O     6.562275    9.478037    1.612089
O     9.072443    8.067706    1.613636
O     9.037037   10.946776    1.612089
O     4.613032   10.375894    3.468621
O     2.567549   12.186532    3.467073
O     5.156641   13.052675    3.467073
O     5.538335    7.892977    4.065805
O    10.956700    7.972753    4.067352
O     8.180832   12.626847    4.067352
O     3.630218    8.587629    1.381569
```

```
O     1.509019   13.931218    1.380022
O     7.197164   13.096416    1.381569
O    -4.111644   11.871192    0.946831
O    -0.000956    9.498504    4.208139
O    -1.662325    9.478037    1.612089
O     0.847843    8.067706    1.613636
O     0.812437   10.946776    1.612089
O    -3.611568   10.375894    3.468621
O    -5.657051   12.186532    3.467073
O    -3.067959   13.052675    3.467073
O    -2.686265    7.892977    4.065805
O     2.732100    7.972753    4.067352
O    -0.043768   12.626847    4.067352
O    -4.594382    8.587629    1.381569
O    -6.715581   13.931218    1.380022
O    -1.027436   13.096416    1.381569
O     8.225356    4.748306    0.946831
O    12.336044    2.375618    4.208139
O    10.674675    2.355151    1.612089
O    13.184843    0.944820    1.613636
O    13.149437    3.823890    1.612089
O     8.725432    3.253009    3.468621
O     6.679949    5.063646    3.467073
O     9.269041    5.929789    3.467073
O     9.650735    0.770091    4.065805
O    15.069100    0.849868    4.067352
O    12.293232    5.503961    4.067352
O     7.742618    1.464744    1.381569
O     5.621419    6.808332    1.380022
O    11.309564    5.973530    1.381569
Ga      4.111444    2.375252    2.390285
Ga      6.370774    0.077427    2.481564
Ga      4.973339    5.477999    2.481564
Ga      0.994332    1.568960    2.481564
Ga      0.000756    4.748658    2.690424
Ga      8.223644    9.500216   12.704867
Ga     10.482974    7.202392   12.796147
Ga      9.085539   12.602964   12.796147
Ga      5.106532    8.693924   12.796147
Ga     4.112956   11.873622   13.005006
Ga    -0.000956    9.500216   12.704867
Ga     2.258374    7.202392   12.796147
Ga     0.860939   12.602964   12.796147
Ga    -3.118068    8.693924   12.796147
Ga    -4.111644   11.873622   13.005006
Ga    12.336044    2.377331   12.704867
Ga    14.595374    0.079506   12.796147
Ga    13.197939    5.480078   12.796147
Ga     9.218932    1.571038   12.796147
Ga     8.225356    4.750737   13.005006
Ga     4.111444    2.377331   12.704867
Ga     6.370774    0.079506   12.796147
Ga     4.973339    5.480078   12.796147
Ga     0.994332    1.571038   12.796147
Ga     0.000756    4.750737   13.005006
Ga     8.223644    9.499177    7.546802
Ga    10.482974    7.201353    7.639629
Ga     9.085539   12.601925    7.639629
Ga     5.106532    8.692885    7.639629
Ga     4.112956   11.872583    7.848489
Ga    -0.000956    9.499177    7.546802
Ga     2.258374    7.201353    7.639629
Ga     0.860939   12.601925    7.639629
Ga    -3.118068    8.692885    7.639629
Ga    -4.111644   11.872583    7.848489
Ga    12.336044    2.376291    7.546802
Ga    14.595374    0.078467    7.639629
Ga    13.197939    5.479039    7.639629
Ga     9.218932    1.569999    7.639629
```

```
Ga     8.225356    4.749697    7.848489
Ga     4.111444    2.376291    7.546802
Ga     6.370774    0.078467    7.639629
Ga     4.973339    5.479039    7.639629
Ga     0.994332    1.569999    7.639629
Ga     0.000756    4.749697    7.848489
Ga     8.223644    9.498138    2.390285
Ga    10.482974    7.200313    2.481564
Ga     9.085539   12.600885    2.481564
Ga     5.106532    8.691845    2.481564
Ga     4.112956   11.871544    2.690424
Ga    -0.000956    9.498138    2.390285
Ga     2.258374    7.200313    2.481564
Ga     0.860939   12.600885    2.481564
Ga    -3.118068    8.691845    2.481564
Ga    -4.111644   11.871544    2.690424
Ga    12.336044    2.375252    2.390285
Ga    14.595374    0.077427    2.481564
Ga    13.197939    5.477999    2.481564
Ga     9.218932    1.568960    2.481564
Ga     8.225356    4.748658    2.690424
Ge     0.000000    0.000001    0.004641
Ge     4.112200    7.124965   10.319223
Ge    12.336800    7.124965   10.319223
Ge     8.224600    0.002080   10.319223
Ge     0.000000    0.002080   10.319223
Ge     4.112200    7.123926    5.161159
Ge    12.336800    7.123926    5.161159
Ge     8.224600    0.001040    5.161159
Ge     0.000000    0.001040    5.161159
Ge     4.112200    7.122887    0.004641
Ge    -4.112400    7.122887    0.004641
Ge     8.224600    0.000001    0.004641
La     3.448574    0.036635    5.063691
La     6.469429    2.969839    5.063691
La     2.420442    4.119473    5.063691
La     7.560774    7.161599   15.378273
La    10.581629   10.094804   15.378273
La     6.532642   11.244438   15.378273
La    -0.663826    7.161599   15.378273
La     2.357029   10.094804   15.378273
La    -1.691958   11.244438   15.378273
La    11.673174    0.038714   15.378273
La    14.694029    2.971918   15.378273
La    10.645042    4.121552   15.378273
La     3.448574    0.038714   15.378273
La     6.469429    2.971918   15.378273
La     2.420442    4.121552   15.378273
La     7.560774    7.160560   10.220208
La    10.581629   10.093764   10.220208
La     6.532642   11.243398   10.221756
La    -0.663826    7.160560   10.220208
La     2.357029   10.093764   10.220208
La    -1.691958   11.243398   10.221756
La    11.673174    0.037674   10.220208
La    14.694029    2.970878   10.220208
La    10.645042    4.120513   10.221756
La     3.448574    0.037674   10.220208
La     6.469429    2.970878   10.220208
La     2.420442    4.120513   10.221756
La     7.560774    7.159521    5.063691
La    10.581629   10.092725    5.063691
La     6.532642   11.242359    5.063691
La    -0.663826    7.159521    5.063691
La     2.357029   10.092725    5.063691
La    -1.691958   11.242359    5.063691
La    11.673174    0.036635    5.063691
La    14.694029    2.969839    5.063691
La    10.645042    4.119473    5.063691
```

```
    &END
    &KIND O
      BASIS_SET SZV-GTH
      POTENTIAL GTH-PBE-q6
    &END KIND
    &KIND La
      BASIS_SET DZVP
      POTENTIAL GTH-PBE-q11
    &END KIND
    &KIND Ge
      BASIS_SET DZVP-MOLOPT-SR-GTH
      POTENTIAL GTH-PBE-q4
    &END KIND
    &KIND Ga
      BASIS_SET DZVP-MOLOPT-SR-GTH
      POTENTIAL GTH-PBE-q13
    &END KIND
  &END SUBSYS
&END FORCE_EVAL
&GLOBAL
  PROJECT tmp
  RUN_TYPE CELL_OPT
  PRINT_LEVEL MEDIUM
  FLUSH_SHOULD_FLUSH
  SAVE_MEM
  &TIMINGS
    THRESHOLD 0.000001
  &END
&END GLOBAL
```

Figure 14: Langasite benchmark CP2K input file. The highlighted sections indicate the parts of the input file that were changed to reduce the runtime and memory requirements down to a manageable level for execution on the Xeon Phi.

Appendix B: CP2K input file for FFT benchmark

This appendix contains the CP2K input file for running the simple FFT benchmark on the Xeon Phi.

```
&GLOBAL
  PRINT_LEVEL MEDIUM
  PROGRAM_NAME TEST
  RUN_TYPE NONE
  &TIMINGS
     THRESHOLD 0.00000000001
  &END
&END GLOBAL
&TEST
  &PW_TRANSFER
    GRID 125 125 125
    N_LOOP 100
  &END
&END
```

Figure 15: fft.inp input file used to compare the performance of FFTW 3.3.3 and MKL 11.1.0 on the Xeon Phi. The grid size used was 125 x 125 x 125 elements.